VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

XML Parser v C++

C++ XML Parser

2020 Jakub Vitásek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Vitásek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: XML Parser v C++
C++ XML Parser

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je naimplementovat parser jazyka XML v C++. Parser bude implementovat přístupy SAX i DOM. Důraz bude kladen na efektivitu. Práce bude splňovat následující body:

1. Důkladný popis jazyka XML.
2. Návrh architektury parseru.
3. Implementace parseru v jazyce C++.
4. Srovnání s existujícími parsery.
5. Tvorba uživatelské dokumentace.

Seznam doporučené odborné literatury:

[1] World Wide Web Consortium. "Extensible markup language (XML) 1.1." (2006).
[2] Lu, Wei, Kenneth Chiu, and Yinfei Pan. "A parallel approach to XML parsing." Proceedings of the 7th IEEE/ACM International Conference on Grid Computing. IEEE Computer Society, 2006.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Lukáš**

Datum zadání: 01.09.2018
Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 15. května 2020 .....

Rád bych poděkoval Ing. Petru Lukášovi, Ph.D. za odbornou pomoc a konzultaci při vytváření této bakalářské práce.

Abstrakt

Cílem této bakalářské práce je návrh a implementace parsera jazyka XML v jazyce C++ s přístupem pomocí rozhraní SAX i DOM. Prioritami při jeho vytváření byla rychlost zpracování XML dokumentu, efektivita při práci s pamětí a jednoduché, uživatelsky přívětivé rozhraní. Teoretická část obsahuje podrobné seznámení s jazykem XML, implementaci parseru, jeho srovnání s již existujícími parsery v různých programovacích jazycích a návod na používání v podobě uživatelské dokumentace.

Klíčová slova: XML, C++, Parser, SAX, DOM

Abstract

The aim of this bachelor thesis is to design and implement XML parser in C++ language with SAX and DOM interface. Priorities while creating parser were speed of XML document processing, memory efficiency and simple user-friendly interface. The teoretical part contains detailed introduction to the XML language, the implementation of the parser, comparison with existing parsers in different programming languages and user manual in form of user documentation.

Keywords: XML, C++, Parser, SAX, DOM

Obsah

Seznam použitých zkratek a symbolů

XML	–	Extensible Markup Language
SGML	–	Standard Generalized Markup Language
W3C	–	World Wide Web Consortium
DTD	–	Document Type Declaration
SAX	–	Simple API for XML
DOM	–	Document Object Model
CSV	–	Comma-separated values

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

XML je značkovací jazyk využívaný jako obecný formát pro uchovávání strukturovaných dat. Je snadno čitelný a upravitelný i pro člověka v libovolném textovém editoru a má velmi jednoduchou syntaxi. Díky tomu, že je rozšiřitelný, je možné v něm vytvářet vlastní formáty a lze s ním popsat téměř jakákoliv data. Používán je nejčastěji pro výměnu dat mezi aplikacemi. Největší nevýhodou u již existujících parserů v jazyce C++ je jejich obtížné použití se špatně udržitelným kódem.

Úkolem této bakalářské práce je prvně důkladně popsat co je to XML, podrobně rozebrat jeho syntaxi a jaké jsou standardizované rozhraní pro zpracování XML dokumentu.

Poté popsat vlastní postup při vývoji parseru. Jako první jsou vyjmenovány důležité nástroje použité při vývoji. Poté následuje návrh architektury, u které je dbáno na to, aby při běhu spotřebovala co nejméně strojového času procesoru a operační paměti, ale zároveň aby byl výsledný kód co nejvíce uživatelsky přívětivý, aby jej byl schopný použít i méně zkušený programátor. Dále je popsána jak probíhala optimalizace parseru. Následuje porovnání naimplementovaného parseru s existujícími implementacemi v různých programovacích jazycích.

Jako poslední je vytvořena uživatelská dokumentace, kde je popsáno jak používat vytvořený parser, jaké obsahuje třídy a metody a jaké výjimky mohou nastat.

2 XML

Extensible Markup Language (XML), neboli v češtině rozšiřitelný značkovací jazyk, který se řadí do skupiny značkovacích jazyků, tedy metajazyků, které označují význam jednotlivých částí dokumentů a nikoliv jejich vzhled. O XML se také hovoří jako o samopopisném jazku, který kromě vlastního dokumentu dokáže popsat i svoji strukturu. To znamená, že je jazyk snadno čitelný jak pro stroj tak i člověka. [1]

Vyvinut a standardizován byl skupinou XML Working Group vzniklou pod záštitou konsorcia W3C roku 1996. XML Working Group předsedal Jon Bosak ze společnosti Sun Microsystems s aktivní účastí skupiny XML Special Interest Group, předešle známou jako SGML Working Group. XML vzniklo jako odvozený jazyk od SGML. Obsahuje pouze tu nejdůležitější množinu jazyka SGML, jelikož se zjistilo, že se v praxi používá jen zlomek jeho možností. [2]

Cílem návrhu XML bylo, aby splňoval následující vlastnosti: [3]

1. *XML musí být snadno použitelné přes internet.*
2. *XML musí podporovat širokou škálu aplikací.*
3. *XML musí být kompatibilní s SGML.*
4. *Musí být snadné napsat programy, které zpracovávají XML dokumentů.*
5. *Počet volitelných funkcí v XML má být udržován na absolutním minimu.*
6. *XML dokumenty by měly být čitelné a přiměřeně jasné.*
7. *Návrh XML by měl být připraven rychle.*
8. *Návrh XML by měl být formální a stručný.*
9. *XML dokumenty by měly být jednoduché na vytvoření.*
10. *Rozmanitost v XML značkách má minimální význam.* (překlad autora)

V XML se již od úplného počátku využívá znaková sada ISO/IEC 10646, což umožňuje, aby mohly dokumenty obsahovat texty v mnoha světových jazycích najednou. Je možné využít i jiné libovolné kódování jako je CP-1250 nebo ISO 8859-2, pokud jsou definované na začátku dokumentu. [4]

2.1 Využití

XML je určeno především pro reprezentaci libovolných datových struktur a výměnu dat mezi aplikacemi. Díky tomu, že je specifikace XML zdarma veřejně přístupná na stránkách konsorcia W3C, může kdokoliv bez problémů implementovat podporu XML do svých aplikací. Díky tomu si tento jazyk získal svou popularitu oproti firemním formátům, které často nemají žádnou

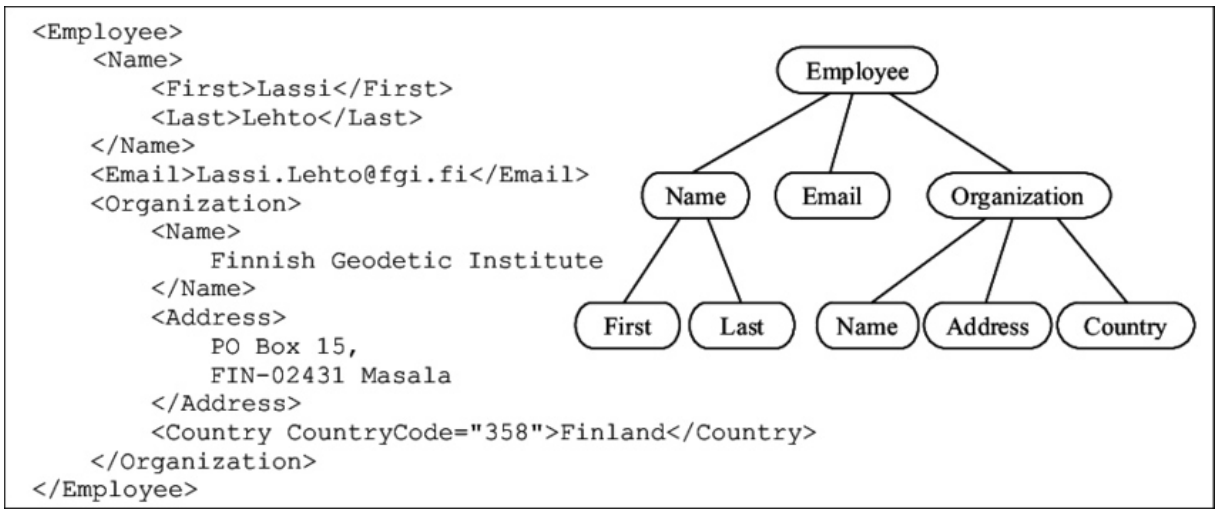
dokumentaci a v porovnání s XML jsou složité. Mezi nejznámější aplikace XML patří XHTML, RSS, SVG a SOAP. [4]

2.2 Verze XML

XML bylo do roku 2006 pouze ve verzi 1.0. Až poté se k němu přidala verze 1.1 se zpětnou kompatibilitou pro verzi 1.0, což znamená, že všechny dokumenty ve staré verzi budou platné i v té nové. Verze 1.1 se od verze 1.0 liší pouze v požadavcích na použité znaky. Ve verzi 1.0 bylo možné v názvech elementů, atributů a procesních instrukcích využívat pouze znaků platných v Unicode 2.0. Celková filozofie jmen se ve verzi 1.1 změnila. Zatímco verze 1.0 poskytuje přesnou definici co musí jména obsahovat, kde cokoliv co není povoleno, je zakázáno, verze 1.1 je navržena naopak tím způsobem, že vše co není z nějakého specifického důvodu zakázáno, je povoleno. [5]

2.3 Syntaxe

XML dokument je tvořen hierarchickou stromovou strukturou tvořenou z jednotlivých uzlů XML. Strom začíná kořenovým elementem a větve stromu jsou tvořeny dalšími prvky XML.



Obrázek 1: Stromová struktura XML. [6]

2.3.1 Tag

Každý tag musí začínat znakem '<' a končit '>'. Všechny tagy musí být párové, tzn. každý počáteční tag musí mít odpovídající ukončovací tag. Jedinou výjimku tvoří prázdné tagy. Počáteční tag se tvoří '<**NAZEV**>', kde mezi '<' a '**NAZEV**' nesmí být mezera. '**NAZEV**' může být tvořen libovolnými alfanumerickými znaky, včetně pomlčky, podtržítka, tečky, a dvojtečky. Nesmí ale začínat pomocí čísla, dvojtečky nebo podtržítka. Tag se také nesmí jmenovat "xml"a to bez ohledu na velikost písmen. Za názvem může být libovolný počet bílých znaků. Ukončovací tag obsahuje navíc '/' před názvem. Název ukončovacího tagu musí být shodný s odpovídajícím

počátečním tagem a rozlišují se malá a velká písmena. Prázdný tag se tvoří stejně jako počáteční tag s jediným rozdílem, že končí '/>' místo pouze '>'. Na výpisu 1 je ukázka počátečního, ukončovacího a prázdného tagu.

```
<weather>
</weather>
<empty/>
```

Výpis 1: Příklad 3 typů tagů.

2.3.2 Bílé znaky

Pro větší přehlednost textu bývají vkládány bílé znaky. Jedná se o mezeru, tabulátor a nový řádek a návrat vozíku. Bílé znaky se rozdělují na významné a nevýznamné. Významné bílé znaky se objevují uvnitř elementu obsahující text, kdy jsou mezery součástí textu. Naopak nevýznamné bílé znaky se objevují na místě, kde je povolen pouze obsah tagu nebo mezi jednotlivými tagy. V této práci budou takovéto znaky ignorovány.

2.3.3 Element

Element je základní stavební blok XML dokumentu. Element může obsahovat text, poznámky, atributy, jiné elementy a další prvky XML. Začátek elementu se značí pomocí počátečního tagu a konec pomocí ukončovacího tagu. Obsah se vkládá mezi počáteční a ukončovací tag. Výjimku tvojí atributy, které se vkládají přímo do počátečního tagu. Element bez obsahu tvoří pouze prázdný tag. Elementy musí být uzavřeny ve správném pořadí. Element otevřený uvnitř jiného elementu, nesmí být uzavřen dříve, než je uzavřen vnější element. Každý XML Dokument musí obsahovat pouze jeden kořenový element. Jedná se o první element v dokumentu. Ukázku elementu "weather"vidíme na výpisu 2.

```
<weather>
...
obsah
...
</weather>
```

Výpis 2: Příklad elementu.

2.3.4 Atribut

Atribut je součástí elementu a popisuje jeho vlastnosti. Element může obsahovat mnoho unikátních atributů. V jednom elementu se může stejný název objevit pouze jednou. Atribut se vkládá za název počátečního nebo prázdného tagu. Tvořen je párem název-hodnota. Název atributu využívá stejná pravidla pojmenování jako název tagu. Hodnota atributu je uzavřena mezi dvojité

nebo jednoduché uvozovky. Atribut se zapisuje ve tvaru '**NAZEV = HODNOTA**'. Ve výpisu 3 lze vidět příklad použití atributu "unit"s hodnotou "Celsius".

```
<temp unit="Celsius">15</temp>
```

Výpis 3: Příklad atributu.

2.3.5 Komentář

Komentář slouží k zahrnutí souvisejících dodatečných informací. Může se objevit kdekoliv v dokumentu, ale nesmí se nořit do sebe. Komentář začíná pomocí '**<!--**' a končí '**-->**'. Uvnitř komentáře může být cokoliv kromě dvou pomlček po sobě. Ve výpisu 4 vidíme ukázku komentáře.

```
<-- Text komentáře -->
```

Výpis 4: Příklad komentáře.

2.3.6 Entita

Entity slouží pro vypsání speciální znaků, které nelze napsat přímo pomocí klávesnice, nebo rezervovaných znaků, které není možné použít v obsahu elementu nebo hodnotě atributu. Každá entita začíná znakem '**&**' a končí '**;**'. Entity se dají zapsat dvěma způsoby. První způsob zápisu je pomocí předem definovaných názvů entit, kterých je v XML celkem 5. Zapisují se '**&NAZEV;**'. Druhý způsob je pomocí číselného odkazu na znak ve znakové sadě. Ten může být jak jak v desítkové tak i hexadecimální soustavě. V desítkové soustavě se zapisuje v podobě '**# číslo**', v případě hexadecimálního zápisu '**#x hex číslo**'. Tabulka 1 obsahuje seznam všech předdefinovaných entit s jejich číselným odkazem v desítkové a hexadecimální soustavě.

Znak	Předdefinovaná entita	Desítkový odkaz	Hexadecimální odkaz
&	&	"	"
"	"	&	&
'	'	'	'
<	<	<	<
>	>	>	>

Tabulka 1: Seznam existujících předdefinovaných entit a jejich číselných odkazů.

2.3.7 CData

CData znamenají v překladu značková data. Slouží pro vložení textu, který obsahuje spoustu značek. Aby nebylo nutné všechny značky nahrazovat příslušnými entitami, CData budou brát všechny znaky jako normální text. CData můžou být vložena kdekoliv uvnitř elementu. Obsah

značkových dat se vkládá mezi sekvenci '<**![CDATA**[' ukončenou sekvencí ']]>'. Na výpisu 5 je vidět ukázka použití značkových dat.

```
<![CDATA[ <element> element co neni interpretovany </element> ]]>
```

Výpis 5: Příklad značkových dat.

2.3.8 Procesní instrukce

Účelem procesních instrukcí je předat instrukce pro aplikace, které budou dokument zpracovávat. Smí se objevit kdekoliv v dokumentu. Začínají '<**?**' a končí '**?**>'. Zapisují se '<**?CÍL INSTRUKCE?**>', kde '**CÍL**' identifikuje, které je instrukce směřována a '**INSTRUKCE**' popisuje informace, které má aplikace zpracovat. Cíl se nesmí jmenovat "xml"a to bez ohledu na velikost písmen. Ukázka procesní instrukce "myapp"je vidět na výpisu 6.

```
<?myapp color="red"?>
```

Výpis 6: Příklad procesní instrukce.

2.3.9 Deklarace

Deklarace XML je speciální procesní instrukce určená pro přípravu XML parseru pro čtení XML dokumentu. Je volitelná, ale pokud se v dokumentu objeví, musí být v dokumentu pouze jedenkrát a to na prvním řádku. '**CÍL**' musí být "xml"a '**INSTRUKCE**' se skládá ze tří volitelných parametrů. Parametry se skládají podle stejných pravidel jako atributy. Deklarace je povinná pouze od verze 1.1. Syntaxi XML deklarace můžeme vidět na výpisu 7. V tabulce 2 se nachází všechny možné parametry deklarace.

```
<?xml version="ČÍSLO VERZE" encoding="KÓDOVÁNÍ" standalone="YES/NO"?>
```

Výpis 7: Syntaxe XML deklarace.

Parametr	Hodnota	Výchozí hodnota
Version	1.0 nebo 1.1	1.0
Encoding	UTF-8, UTF-16, ISO-8859-1...	UTF-8
Standalone	yes nebo no	yes

Tabulka 2: Parametry XML deklarace.

- **Version** - Definuje specifikaci standardu verze XML.
- **Encoding** - Určuje užité kódování znaků v dokumentu.
- **Standalone** - Sděluje parseru, jestli dokument odkazuje na externí DTD.

2.3.10 Jmenný prostor

Jmenný prostor je sada jedinečných jmen, identifikována pomocí URI, díky nimž je možné přiřadit elementy a atributy do skupiny. Toho se využívá při míchání různých XML dokumentů z různých XML aplikací, kde vznikají kolize názvů elementů nebo atributů. Deklarován je pomocí rezervovaného atributu, kdy název atributu se musí jmenovat '**xmlns**' nebo začínat '**xmlns:**' následovaný prefixem jmenného prostoru. Hodnota atributu obsahuje identifikátor jmenného prostoru. Jmenný prostor ovlivňuje pouze určitou oblast dokumentu: element obsahující jeho deklaraci a všechny jeho potomky. Výpis 8 ukazuje příklad použití jmenného prostoru s identifikátorem "www.vsb.cz"a prefixem "vsb".

```
<vsb:student xmlns:vsb = "www.vsb.cz">
   <vsb:firstname>Jakub</vsb:firstname>
   <vsb:lastname>Vitasek</vsb:lastname>
   <vsb:height vsb:unit="cm">178</vsb:height>
</vsb:student>
```

Výpis 8: Příklad použití jmenného prostoru.

2.3.11 DTD

Document type definition (DTD), neboli definice typu dokumentu, umožňuje definovat strukturu dokumentu a tím pomocí parseru hlídat její správnost. Určuje které elementy a atributy můžeme v dokumentu použít a v jakém vzájemném vztahu mohou být použity. DTD se musí nacházet před kořenovým elementem v dokumentu. Rozděluje se na 2 typy, podle toho, kde se definice nachází:

- **Interní DTD** - definován uvnitř XML dokumentu
- **Externí DTD** - definován v samostatném souboru

DTD začíná pomocí sekvence '**<!DOCTYPE**', která je následována **kořenovým elementem**, **DTD identifikátorem** a ukončena znakem '>'. Výpis 9 obsahuje obecnou syntaxi DTD.

```
<!DOCTYPE element DTD_identifikátor>
```

Výpis 9: Syntaxe DTD.

V případě interního DTD obsahuje identifikátor seznam deklarací uzavřených do hranatých závorek. DTD může obsahovat 4 typy deklarací:

- deklaraci elementů,
- deklaraci atributů,
- deklaraci entit,

- deklaraci notací.

Na výpisu 10 je ukázka interního DTD.

```
<!DOCTYPE contact [
   <!ELEMENT contact (name, phone)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT phone (#PCDATA)>

   <contact>
      <name>Jakub</name>
      <phone>123 456 789</phone>
   </contact>
]>
```

Výpis 10: Příklad interního DTD.

Externí DTD, jak již bylo řečeno, odkazuje na deklaraci elementů mimo XML soubor. Musí se jednat o platný soubor s příponou '**.dtd**'. XML deklarace musí být parametr '**standalone**' nastavený na '**no**'. Ukázku použití externího DTD lze vidět ve výpisu 11.

```
<!DOCTYPE element SYSTEM "file.dtd">
```

Výpis 11: Příklad externího DTD.

2.4 Standardizovaná rozhraní

Jelikož jsou XML dokumenty obyčejné textové soubory, je pro programátora čtení a zápis XML velice snadný. Pro čtení jsou k dispozici programátorovi k dispozici již existující parsery ve většině dnes běžně používaných programovacích jazycích. Parsery mají za úkol zpřístupnit obsah dokumentu v jednoduché podobě a zároveň zkontrolovat syntaktickou správnost dokumentu.[7]

Navzdory tomu, že v cílech při návrhu XML je uvedeno, *"Musí být snadné napsat programy, které zpracovávají XML dokumentů."*, neexistuje téměř žádná informace, jak by mělo XML zpracovávat [8]. I přes to bylo několik přístupů pro zpracování XML vyvinuto, z nichž byly některé standardizovány. Mezi ty nejvíce používané patří SAX a DOM.

2.4.1 SAX

Rozhraní SAX (Simple API for XML) funguje na principu řízení pomocí událostí, kdy dokument je čten sériově a pokaždé, co narazí na nějaký prvek dokumentu, jako je začátek elementu, text, komentář, atd., vyvolá určitou registrovanou metodu, které jsou do parametrů předána všechna potřebná data.

Tento přístup je velmi rychlý, efektivní, jednoduchý na implementaci a využívá pouze malou část paměti, jelikož je dokument zpracováván postupně a není potřeba ho načítat do paměti celý. Díky tomu se hodí na čtení velkých dokumentů Je však velmi obtížné ho použít pro získání dat z náhodných pozic v dokumentu, jelikož aplikace musí sledovat, která část dokumentu se zrovna zpracovává.

2.4.2 Pull parsing

Pull parsing je vylepšené rozhraní SAX, kdy prvky dokumentu jsou čteny sériově pomocí návrhového vzoru iterátor. Je vytvořen iterátor, který sekvenčně prochází jednotlivé prvky dokumentu. Aplikace pak má možnost u každé položky iterátoru zjistit, o jaký typ prvku dokumentu se jedná, získat všechna potřebná data a posunout se k dalšímu prvku. Kód využívající přístup pull parsing je oproti SAX jednodušší na pochopení a lépe udržovatelný. [8]

2.4.3 DOM

Rozhraní DOM (Document Object Model) načítá celý dokument do stromové hierarchické struktury, kde každému prvku dokumentu odpovídá právě jeden uzel stromu. DOM model může být vytvořen parserem, nebo manuálně uživatelem. Rozhraní obsahuje funkce umožnující procházení celého stromu, modifikování, mazání a vytváření jednotlivých uzlů.

Jelikož je před přístupem k datům potřeba načíst celý dokument do paměti ve formě stromové struktury, je velmi náročný na využití paměti. Jelikož naopak od rozhraní SAX není potřeba procházet dokument sekvenčně od začátku do konce, je možné se v něm pohybovat náhodně podle potřeby a je velmi efektivní pro komplexnější operace s dokumentem.

2.5 Existující řešení

Téměř každý dnes běžně používaný programovací jazyk má k dipozici knihovnu pro zpracování XML dokumentu. Mezi nejznámější parser v jazyce C++ patří **Xerces**, který nabízí rozhraní SAX, DOM, ale i méně známý StAX. Xerces je k dispozici i v jazycích Java a Perl. V jazyce .NET C# je nejznámější zástupce **XmlReader** s rozhraním pull parsing a **XmlDocument** s rozhraním DOM. Jsou velmi populární hlavně díky jejich jednoduchému používání a velké rychlosti. V jazyce Java je poskytováno parsování XML pomocí přístupu SAX s třídou **javax.xml.parsers.SAXParser** nebo přístupu DOM s třídou **javax.xml.parsers.DocumentBuilder**.

3 Implementace parseru

V této kapitole popíšeme nástroje použité při vývoji parseru a následně jeho implementaci. Cílem při jeho vývoji je snadnost použití a efektivita při načítání dokumentu. Naopak není požadováno kontrolování struktury pomocí DTD a použití jmenného prostoru.

3.1 Nástroje pro vývoj

Celá práce byla programována a testována na systému Windows 10 v prostředí Visual Studio 2019. Kompilátor byl použit výchozí MSVC zabudovaný ve Visual Studiu. Pro profilování kódu byl využit nejen vestavěný profiler ve Visual Studiu, ale také program Very Sleepy.

3.1.1 C++

C++ je multiparadigmatický programovací jazyk, vyvinutý Bjarnem Stroustrupem roku 1985 jako rozšíření jazyka C. Podporuje paradigma procedurálního programování, objektově orientovaného a generického programování a zároveň poskytuje zázemí pro nízkoúrovňovou manipulaci s pamětí. Jedná se o kompilovaný jazyk, navržený pro programování velkých systémů, ale také zdrojově omezených systémů s důrazem na efektivitu. Jazyk je standardizován Mezinárodní Organizací pro Normalizaci (ISO) s nejnovější verzí C++17 v Prosinci 2017. [9]

3.1.2 Visual Studio

Visual Studio je integrované vývojové prostředí, vyvinuté společností Microsoft. Slouží k vývoji konzolových aplikací, aplikací s grafickým rozhraním, webovým aplikacím, mobilním aplikacím a dalším. Visual Studio obsahuje editor kódu podporující IntelliSense a refaktorování. Vestavěný debugger pracuje jak na úrovni kódu, tak na úrovni stroje. Jeho funkčnost lze vylepšovat pomocí rozšíření, jako je například podpora verzovacích systémů. Podporuje více jak 36 programovací jazyků prostřednictvím jazykových služeb, což umožňuje, aby editor kódu a debugger podporoval téměř jakýkoliv programovací jazyk. Mezi vestavěné jazyky patří C++, C# a VB.NET. První verze programu byla vydaná v roce 1997. Nejnovější verze nese označení Visual Studio 2019. Vychází ve třech edicích: Community, Professional, Enterprise. [10]

3.1.3 Microsoft Visual C++

Microsoft Visual C++, také často označovaný jako MSVC nebo VC++, je vývojové prostředí společnosti Microsoft pro jazyky C a C++. MSVC je proprietární software nabízející nástroje pro vývoj a ladění kódu v jazyce C++. Původně se jednalo o samostatný produkt, později se však stal součástí Visual Studia. [11]

3.1.4 Very Sleepy

Jedná se o nástroj pro profilování kódu v jazyce C nebo C++ pro systém Windows. Podporuje jakoukoliv nativní aplikaci pro Windows, pokud splňuje standard ladících informací PDB nebo DWARF2. Podporované jsou jak 32-bitové, tak i 64-bitové systémy. Podporuje kompilátory gcc/mingw tak i MSVC. Výsledek profilování zobrazuje v přehledném grafu jednotlivých volání a výsledek lze exportovat do CSV formátu. [12]

3.2 Návrh architektury parseru

Celý projekt se rozděluje na 2 části. První část projektu má za úkol sériově přečíst XML dokument ze souboru a po částech ho postupně zprostředkovat pomocí uživatelsky přívětivého rozhraní. Druhá část umožní vytvořit v operační paměti hierarchickou strukturu dokumentu a pomocí první části dokáže tuto strukturu naplnit. Oba části budou součásti jmenného prostoru `VsbFeiXml`.

3.2.1 XmlReader

Pro první část projektu nám poslouží třída `XmlReader`. Tato třída umožní sekvenční čtení XML souboru s rozhraním `pull parsing`. Návrh této třídy byl inspirován v rámci `.NET Frameworku`. Využívá návrhového vzoru iterátor, kdy je volána metoda `read()` dokud parser nenarazí na konec souboru. Uživatel je pak schopný získat informace o prvku na který parser narazil pomocí volání metod `getName()`, `getValue()` a `getNodeType()`. Při zpracování dokumentu se striktně řídí specifikací standardu W3C ve verzi 1.1 [5]. Parser předpokládá, že vstupní XML kódovaný v UTF-8.

Než budeme moct soubor zpracovávat, tak ho musíme první načíst do operační paměti. Aby bylo možné zpracovávat i opravdu velké soubory na počítači s omezenou velikostí RAM, je důležité, aby se soubor nenačítal do paměti celý naráz. Soubor se proto bude číst po částech zvaných `chunky`. Jakmile parser zpracuje jednu část, zažádá si o další chunk, pokud je k dispozici. Velikost jednoho chunku je 16384 bytů. Pro čtení ze souboru využívá třídu `ifstream` v binárním módu. Pokud soubor obsahuje BOM, je zkontrolován a přeskočen. UTF-8 používá pouze pořadí bytů big-endian, takže je BOM v případě UTF-8 nepovinný.

Soubor se bude vyhodnocovat znak po znaku. Do této chvíle je soubor rozdělen na jednotlivé byty, ale jelikož v kódování UTF-8 může mít jeden znak až 4 byty, je potřeba jednotlivé znaky první dekódovat. Kódování a dekódování znaků z UTF-8 řeší parser bez použití externích knihoven. Z prvního bytu znaku zjistí potřebný počet bytů pro daný znak, získá zbylé byty a pomocí bitových operací převede znak na UTF-32, který je totožný s unicode hodnotou znaku, kterou lze poté použít zpracovávání XML dokumentu. Výpis 12 obsahuje metodu pro převod UTF-8 znaku na UTF-32.

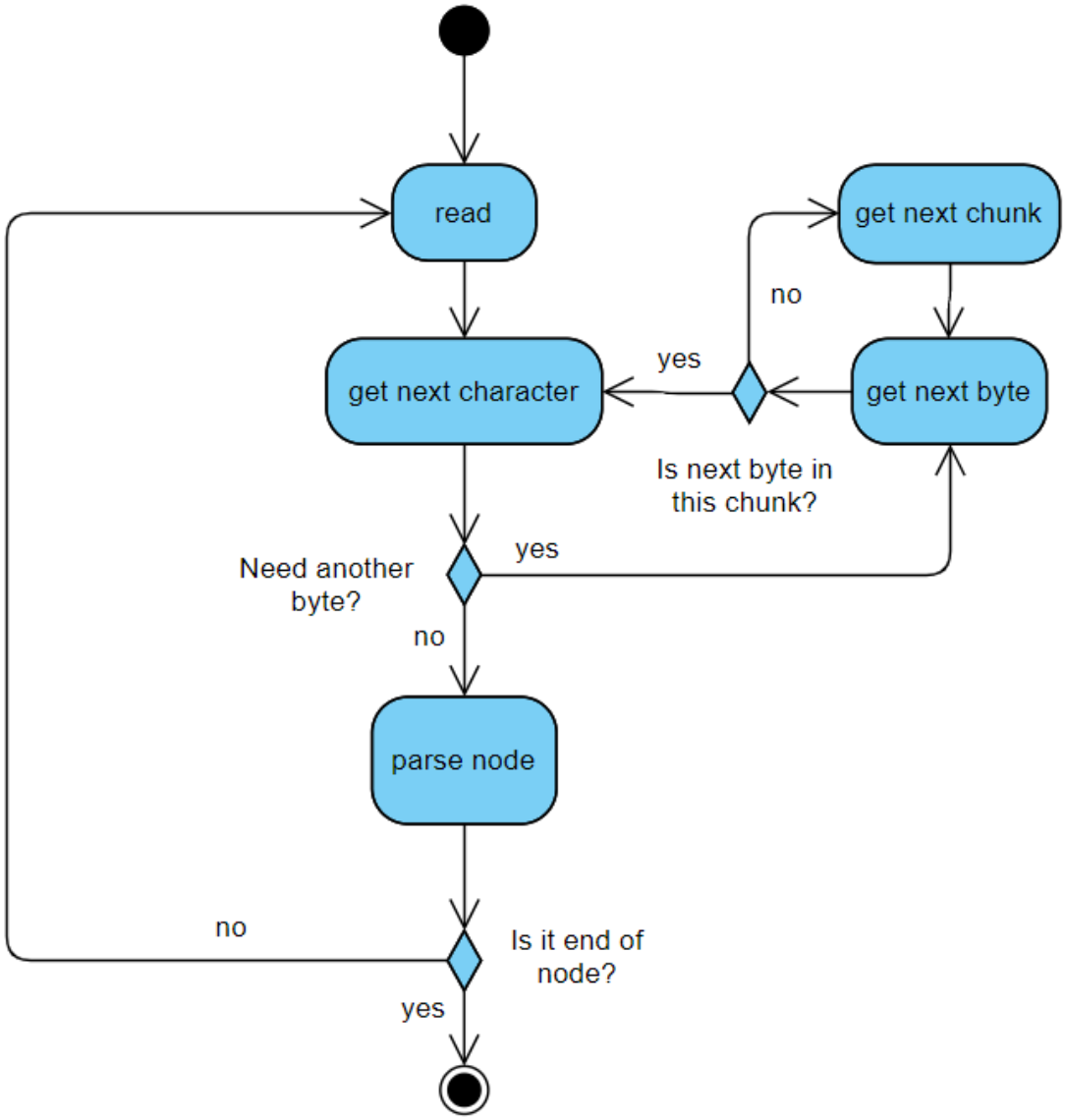
```
bool VsbFeiXml::XmlReader::convertUTF8ToUTF32(Utf8Char* t_input, uint32_t*
    t_output)
{
    static const uint8_t BYTE_MASK = 0x3F;
    static const uint8_t FIRST_BYTE_MASK[5] = { 0, 0xFF, 0x1F, 0xF, 0x7 };

    *t_output = 0;

    switch (t_input->m_length) {
    case 1:
        *t_output = static_cast<uint32_t>(t_input->m_bytes[0]);
        break;
    case 2:
        *t_output |= (BYTE_MASK & t_input->m_bytes[1]);
        *t_output |= ((FIRST_BYTE_MASK[t_input->m_length] & t_input->m_bytes
            [0]) << 6);
        break;
    case 3:
        *t_output |= (BYTE_MASK & t_input->m_bytes[2]);
        *t_output |= (BYTE_MASK & t_input->m_bytes[1]) << 6;
        *t_output |= ((FIRST_BYTE_MASK[t_input->m_length] & t_input->m_bytes
            [0]) << 12);
        break;
    case 4:
        *t_output |= (BYTE_MASK & t_input->m_bytes[3]);
        *t_output |= (BYTE_MASK & t_input->m_bytes[2]) << 6;
        *t_output |= (BYTE_MASK & t_input->m_bytes[1]) << 12;
        *t_output |= ((FIRST_BYTE_MASK[t_input->m_length] & t_input->m_bytes
            [0]) << 18);
        break;
    default:
        return false;
        break;
    }
    return true;
}
```

Výpis 12: Metoda pro převod UTF-8 znaku do UTF-32 kódování.

Když je úspěšně hotové čtení znak po znaku, je na řadě implementace metody `read()`, která bude vracet jednotlivé uzly dokumentu. Parser si bude pamatovat stav, co za uzel zrovna zpracovává. Stav je uložen jako výčtový typ. Podle stavu bude znak po znaku zpracovávat uzel a kontrolovat jeho syntaktickou správnost dokud nedojde na konec uzlu nebo souboru. Obrázek 2 graficky znázorňuje proces zpracovávání jednoho uzlu.



Obrázek 2: Diagram aktivit metody read().

V metodě pro zpracování jednotlivého znaku daného typu uzlu se určuje zda je znak platným znakem podle syntaxe XML standardu W3C. Pokud ano, určí se, jak s daným znakem naložit, jestli je to nevýznamný bílý znak, část názvu tagu a nebo například konec tagu a podobně. Pro ukládání názvů a hodnot je využito třídy `Vector`, která uchovává na heapu pole bytů a umožňuje

s polem provádět snadnou manipulaci. Názvy a hodnoty jsou ukládány již v kódování UTF-8, aby nebylo potřeba poté celý řetězec kódovat z UTF-32 do UTF-8. Výpis 13 je příkladem takové metody. Metoda slouží pro zpracování znaku textového uzlu. Zjistí, jestli znak není začátkem nového elementu, pokud ano, přepne stav na zpracovávání elementu a vrátí metodě `read()`, jestli byl zpracován nový textový uzel, nebo se jednalo pouze o nevýznamné bílé znaky. Pokud se nejedná o začátek elementu, zjistí jestli se nejedná o entitu. Pokud se nejedná ani o entitu, znamená to, že znak patří do textového uzlu. Metoda `read()` tak může zpracovávat další znak.

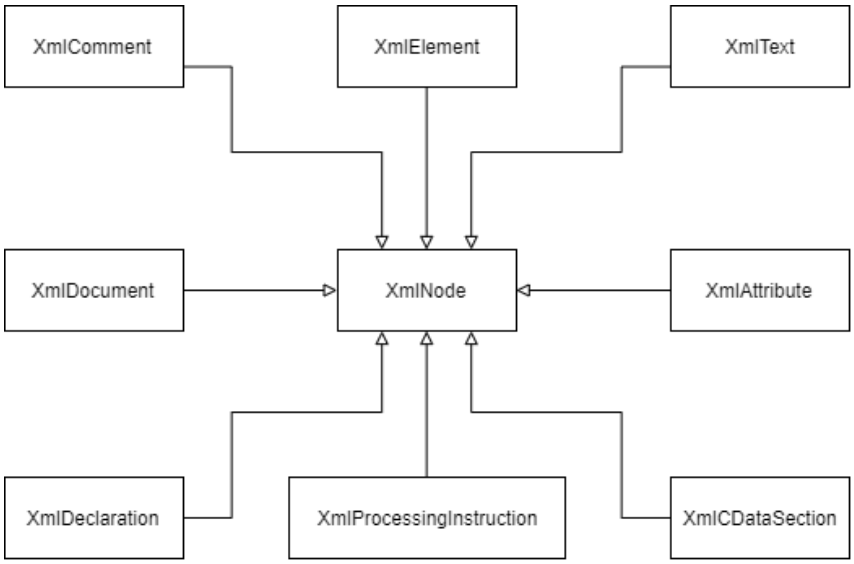
```
bool VsbFeiXml::XmlReader::parseText()
{
    if (m_encodedCharacter == '<') {
        if (m_valueData.size() != 0) {
            m_parserState = ParserState::PARSING_ELEMENT;
            if (isAllWhitespaces(m_valueData)) {
                m_valueData.clear();
                return false;
            }
            m_nodeType = XmlNodeType::TEXT;
            m_valueData.emplace_back('\0');
            return true;
        }
        m_parserState = ParserState::PARSING_ELEMENT;
    }
    else if (m_encodedCharacter == '&') {
        m_parserStateBackup = m_parserState;
        m_parserState = ParserState::PARSING_REF;
    }
    else {
        appendToData(m_valueData, m_decodedCharacter);
    }
    return false;
}
```

Výpis 13: Metoda pro zpracování znaku textového uzlu.

Jelikož Vector při vkládání nových položek pokaždé alokuje nové pole a původní kopíruje do nového, má proto Vector s hodnotou rezervováno 16384 bytů a Vector s názvem má 1024 bytů. Ty se rezervují již v konstruktoru.

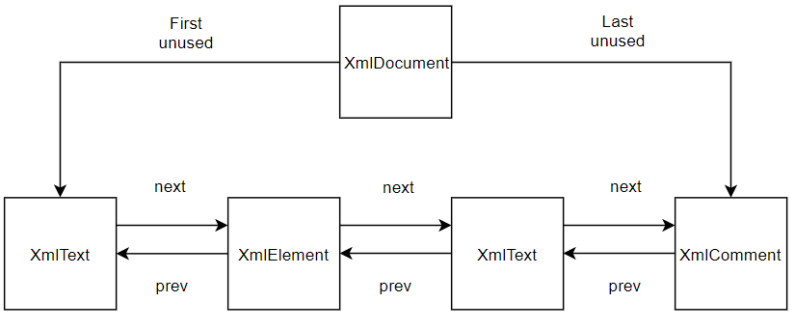
3.2.2 XmlDocument

Druhou část projektu tvojí knihovna `XmlDocument`, která umožnuje načtení XML dokumentu do stromové hierarchické struktury v operační paměti, kde každý prvek XML dokumentu je reprezentován jako jeden uzel stromu. Každý typ uzlu je reprezentován vlastní třídou. Základem všech tříd je abstraktní třída `XmlNode`, která poskytuje metody pro procházení stromu, mazání a editaci jednotlivých uzlů. Všechny třídy tedy ze této třídy dědí. Na obrázku 3 lze vidět hierarchie použitých tříd.



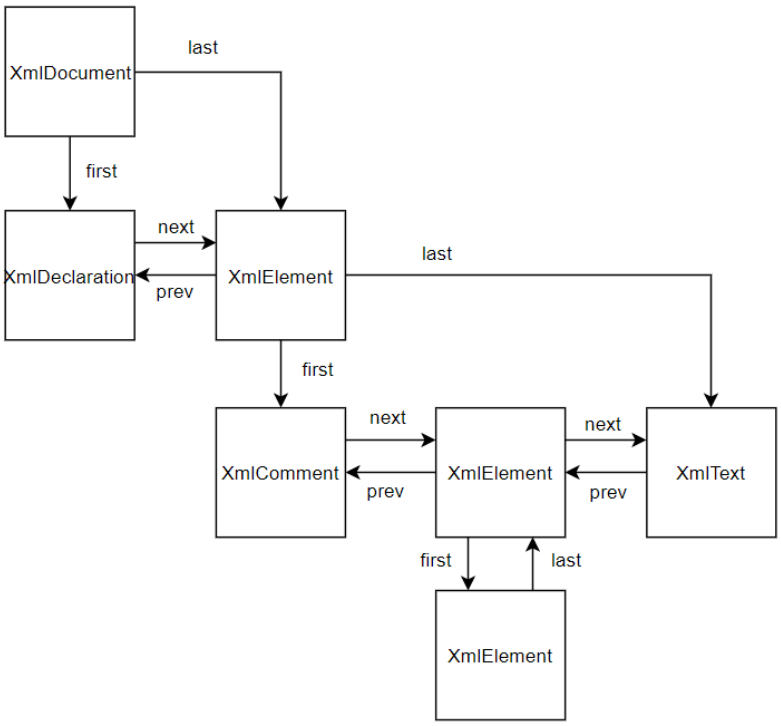
Obrázek 3: Třídní diagram knihovny XmlDocument.

Všechny třídy kromě `XmlDocument` mají privátní konstruktor i destruktor. Je to z důvodu aby předcházelo nečekaným únikům paměti. Z toho důvodu se všechny uzly musí vytvářet pomocí metod `createXXX()` na instanci třídy `XmlDocument`, kde se následně uloží jejich ukazatel do dvoucestného seznamu všech ještě nepoužitých objektů vytvořených pomocí daného dokumentu. Pokud se objekt přidělí některému z uzlů pomocí metody `appendChild(node : XmlNode*)`, je z tohoto seznamu odebrán. Pokud je potřeba uzel vymazat, je potřeba zavolat metodu `removeChild(node : XmlNode*)` na objekt rodiče uzlu. Pokud je vymazán uzel s potomky, destruktor se postará o jejich automatické vymazání. Třídy jsou navzájem označené jako tzn. přátelské třídy. To jim umožňuje přistupovat k privátním členům třídy. V našem případě je to využito pro volání privátních konstruktorů a destruktorů. Na obrázku 4 je struktura dvoucestného seznamu nepoužitých objektů.



Obrázek 4: Dvoucestný seznam nepoužitých objektů.

Pro ukládání potomků uzlů jsem znovu zvolil dvoucestný seznam. Je mnohem efektivnější oproti klasického pole v případě libovolného rozšiřování. Seznam není nutné oproti poli přealokovávat při každém novém vložení nebo vymazání. Uzel si uchovává prvního a posledního potomka. Potomci potom mezi sebou uchovávají další a předcházející sourozenecký uzel. Díky tomu vznikne hierarchický strom jako na obrázku 5.



Obrázek 5: Stromová struktura uzlů složená z dvoucestných seznamů.

Atributy jsou v elementu uloženy do vlastního jednocestného seznamu. V jejich případě není

potřeba zpětné iterace, jelikož je nutné projít celý seznam jen pro vyhledání názvu. Navíc je důležité, aby se nepletly s ostatními uzly, které může element mít.

Dokument lze také načíst pomocí funkce `open(filename : string)`, která použije knihovnu `XmlReader`, která postupně sestaví celou stromovou strukturu dokumentu. Pomocí funkce `save(filename : string)` lze uložit celý dokument do souboru. Dokument se uloží v pořadí uzlů v jakém jsou ve stromové struktuře. V textech, CDatách, komentářích a XML deklaracích jsou vymazány počáteční a končící bílé znaky. Každý uzel je umístěn na nový řádek s odsazením podle úrovně vnoření. Počáteční vnoření pro XML deklaraci a element je 0. Při každém vnoření se přičítá 1. Ve výpisu 14 je příklad souboru vytvořeného pomocí této metody s nejvyšším vnořením na úrovni 2.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<test a="b">
   <![CDATA[characters with markup]]>
   abcčč&#38;&amp;
   <?mso-application progid="Excel.Sheet"?>
   <!--Your comment-->
   <node>
      text
   </node>
   <empty c="d" e="f"/>
</test>
```

Výpis 14: Příklad výpisu souboru vytvořeného pomocí metody save().

3.3 Optimalizace

Aby byl kód co nejvíce optimalizovaný je důležité ho profilovat. Profilování odhalí, které volání v k kódu zabralo nejvíce procesorového času. K této činnosti byl použit program Very Sleepy. Díky tomu bylo možné pomocí nahrazení velmi neefektivního `std::endl` za `\n` ušetřit až 80% času při ukládání XML dokumentu do souboru. Velkou část procesorového času také stojí alokace paměti na heapu. Ke sledování veškeré této alokace bylo využito přetížení operátoru new jako je vidět ve výpisu 15.

```
void* operator new(size_t size)
{
   std::cout << "Allocating " << size << " bytes\n";

   return malloc(size);
}
```

Výpis 15: Sledování alokace na heapu pomocí přetížení operátoru new.

3.4 Srovnání s existujícími parsery

Aby bylo možné zjistit reálnou efektivitu vytvořeného parseru, je důležité ho porovnat s již vytvořenými řešeními. Z toho důvodu bylo provedeno měření rychlosti zpracování xml souboru a porovnáno s často používanými konkurenčními parsery v Jazyce C++, C# a Java. Měření bylo provedeno na počítači s operačním systémem Windows 10 s procesorem Intel® Core™ i7-2700K přetaktovaným na 4.3 Ghz. Jak data, tak program byly uloženy na pevném disku s rozhraním SATA 3.0 s rychlostí 7200 otáčet za minutu a vyrovnávací pamětí 32 MB. Přesnost měření je na jednotky milisekund. Jelikož při měření dochází při několikanásobném spuštění k optimalizaci a cachování samotným systémem, je každé měření spuštěno celkem 5 krát a zapsána je pouze nejnižší hodnota. V rámci srovnávání byly použity různé dokumenty určené k testování parserů. Seznam dokumentů včetně jejich velikosti v kB je v tabulce 3. Srovnání se podle rozhraní parseru dělí na 2 části: SAX a DOM.

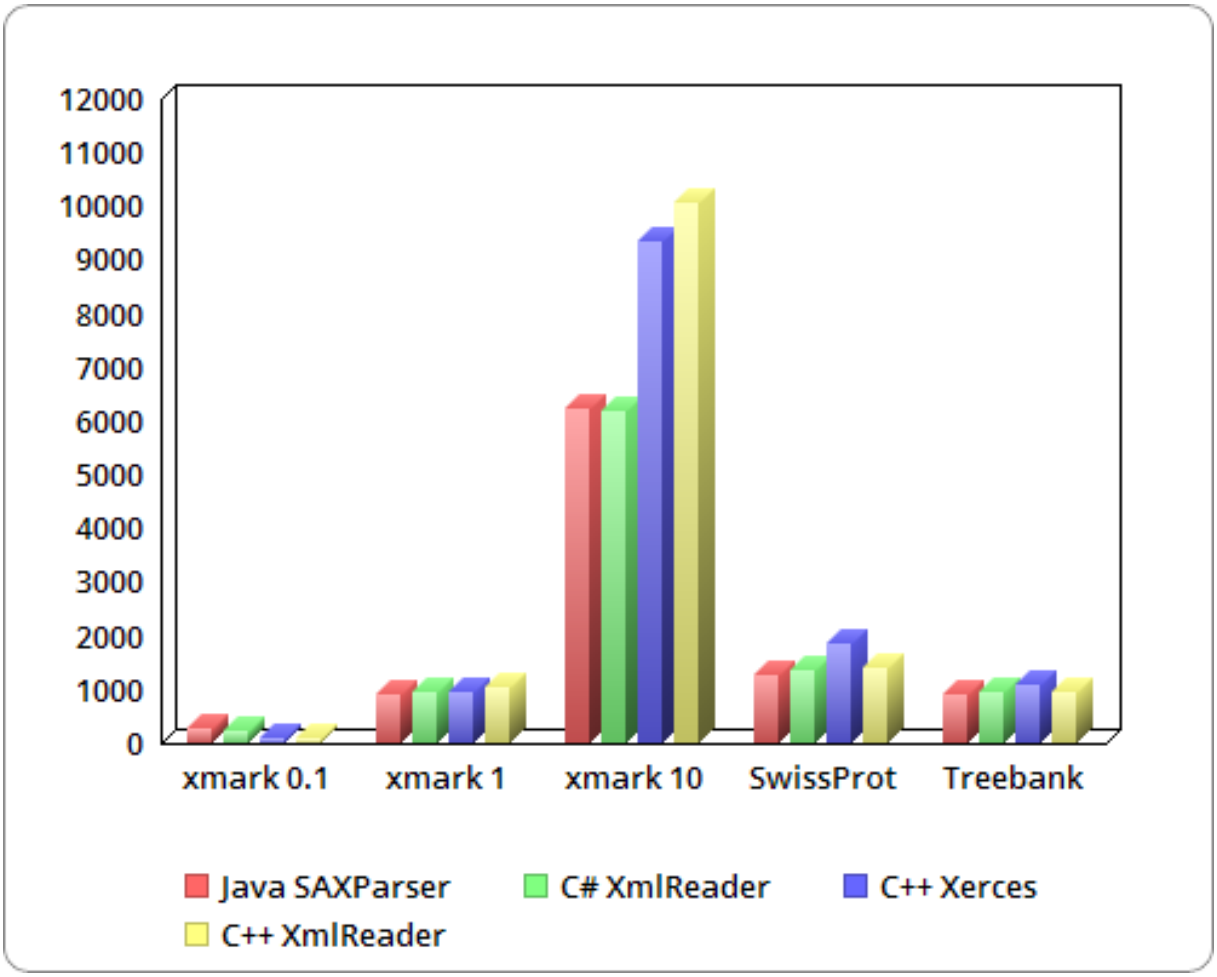
XML dokument	Velikost v kB
XMark /f 0.1	11 597
TreeBank	84 065
SwissProt	112 130
XMark /f 1	115 775
XMark /f 10	1 164 723

Tabulka 3: Seznam XML dokumentů použitých při porovnání parserů.

3.4.1 SAX

Rychlost vytvořeného SAX parseru byla porovnána vůči `Xerces SAX2` v jazyce C++, stejnojmenném `XmlReader` v jazyce C# a `SAXParser` v jazyce Java. Každý parser měl za úkol přečíst celý XML dokument a spočítat všechny výskyty počátečního tagu v dokumentu. Čas se začal počítat od inicializace třídy parseru a přestal počítat po projetí celého dokumentu.

Na obrázku 6 lze vidět graf, zobrazující dobu trvání zpracování každého XML dokumentu jednotlivými parsery. Menší hodnota v tomto grafu znamená lepší výsledek. V případě nejmenšího XML dokumentu `XMark /f 0.1` ho náš parser dokázal zpracovat v polovičním čase oproti těm v jazyce Java a c#. Dokumenty `XMark /f 1`, `SwissProt` a `Treebank` zpracovaly všechny parsery s menšími odchylky téměř ve stejném čase. Již zde lze ale pozorovat lehký nárůst času v případě `C++ Xerces SAX2`. Největší odchylka nastává v případě dokumentu `XMark /f 10`, kde má náš parser společně s `C++ Xerces SAX2` oproti `Java SAXParseru` a `C# XmlReaderu` víc jak 50% nárůst času. Srovnání všech parserů s přesnými časy je v tabulce 4.

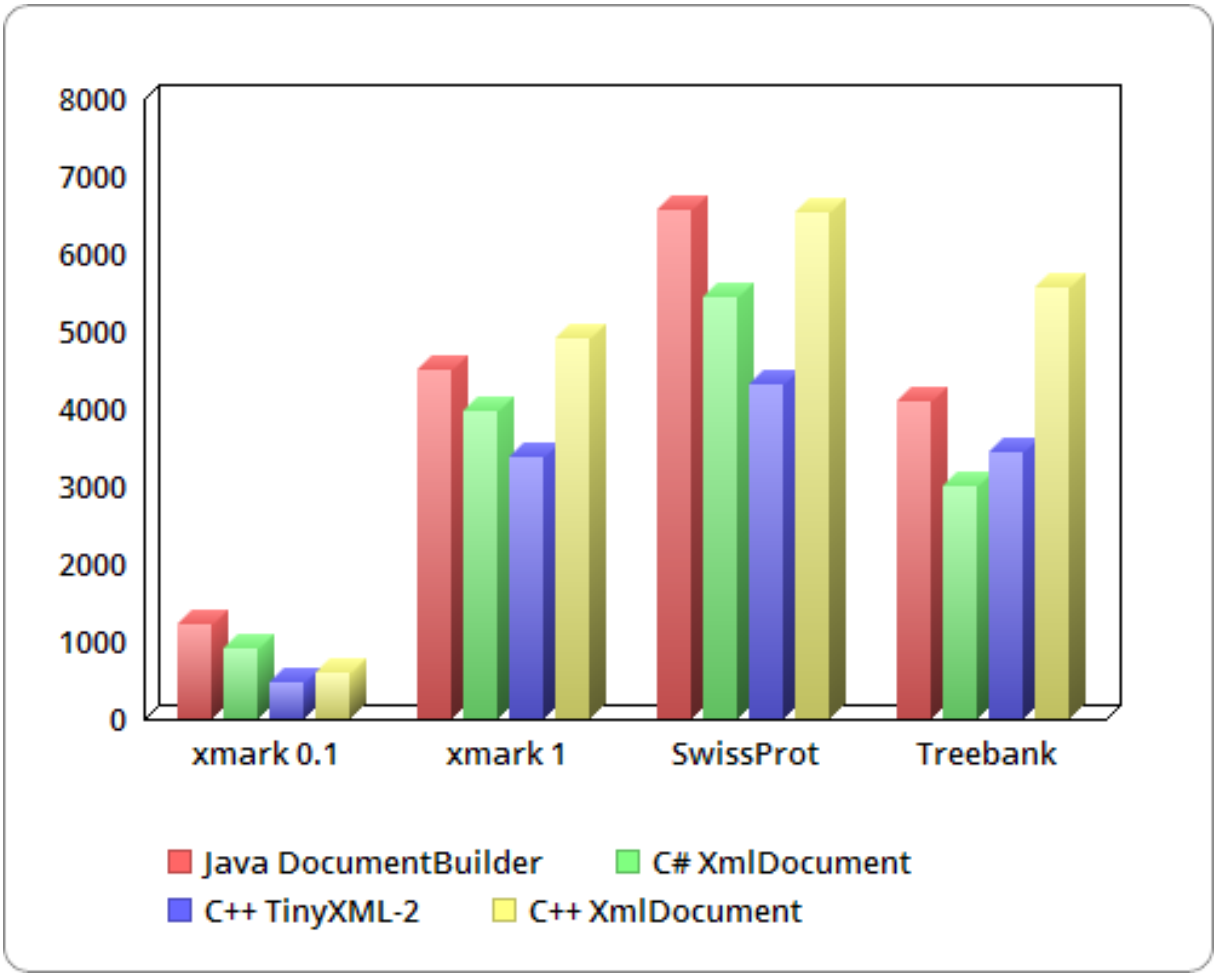


Obrázek 6: Graf porovnání SAX parserů.

3.4.2 DOM

Pro porovnání vytvořeného DOM parseru byl vybrán TinyXML-2 parser v jazyce C++, který se mu svými vlastnostmi nejvíce blíží. Dále byli také vybráni 2 zástupci z ostatních jazyků, kterými jsou `XmlDocument` v jazyce C# a `DocumentBuilder` v jazyce Java. V tomto případě měl každý parser za úkol načíst dokumentu do paměti a poté ho celý uložit do souboru. Čas se začal počítat od inicializace třídy parseru a přestal počítat po uložení souboru. Kvůli paměťové náročnosti byl z tohoto testu vyjmut dokument `XMark /f 10`.

Na obrázku 7 lze vidět graf, zobrazující dobu trvání zpracování každého XML dokumentu jednotlivými parsery. Menší hodnota v tomto grafu znamená lepší výsledek. Zpracování nejmenšího dokumentu trvalo našemu parseru o 50% méně času oproti nejpomalejšímu parseru `Java DocumentBuilder`. Naopak oproti `C++ TinyXML-2`, který to zvládl nejrychleji, zaostává o 22%. V případě dokumentů `XMark /f 1` a `SwissProt` je náš parser o 40-50% pomalejší než nejrychlejší parser. V posledním dokumentu `Treebank` je náš parser až o 85% pomalejší oproti nejrychlejšímu `C# XmlDocument`. Srovnání všech parserů s přesnými časy je v tabulce 5.



Obrázek 7: Graf porovnání DOM parserů.

4 Uživatelská dokumentace

4.1 XmlReader

Třída XmlReader slouží jako jednoduchý, malý parser XML. Využívá standardizované rozhraní pull-parsing a je šetrný k využívání paměti, takže je vhodný ke čtení velkých XML souborů. Nepodporuje DTD, takže není schopný kontrolovat strukturu XML dokumentu. XmlReader je navrhnut tak, aby zvládl přečíst dokument v co nejkratším čase a zároveň aby byl jednoduchý a velmi rychle naučitelný a jeho kód byl lehce udržovatelný. Pro použití třídy je potřeba přesunout soubory **XmlReader.h** a **XmlReader.cpp** do složky se zdrojovými soubory a přidat hlavičkový soubor tam, kde bude používán. Výpisu 16 obsahuje přidání potřebného hlavičkového souboru a následné vytvoření instance třídy XmlReader na stacku. Konstruktor třídy XmlReader má pouze jeden povinný argument **url**, datového typu *string*, který značí cestu k souboru XML.

```
#include "XmlReader.h"

VsbFeiXml::XmlReader reader = VsbFeiXml::XmlReader("xmark1.xml");
```

Výpis 16: Vytvoření instance třídy XmlReader.

Pro práci s třídou poté stačí pouze 5 základních metod:

- **read**() : *bool* - Slouží jako signál pro parser, aby přečetl další prvek XML dokumentu. Pokud na nějaký narazí, vrací *true*, v opačném případě vrací *false*. Pokud nalezne prvek, uloží jeho typ a buď jeho název, hodnotu nebo popřípadě obojí. Pro získání těchto dat slouží následující metody.
- **getNodeType**() : *XmlNodeType* - Vrací typ XML prvku.
- **getName**() : *vector<char>* - Vrací název prvku.
- **getValue**() : *vector<char>* - Vrací hodnotu prvku.
- **setReferenceParsing**(state : *bool*) : *void* - Nastavuje, jestli bude parser dekódovat entitu do znaku v UTF-8 kódováním, nebo ji pouze překopíruje. Výchozí nastavení je true.

XmlNodeType je výčtový typ, který může v případě této třídy nabývat následujících hodnot:

- **START_ELEMENT** - V případě nalezení počátečního tagu nebo začátku prázdného tagu, je získán název tagu.
- **END_ELEMENT** - V případě nalezení ukončovacího tagu nebo konce prázdného tagu, je získán název tagu.
- **END_EMPTY_ELEMENT** - V případě nalezení konce prázdného tagu.

- **TEXT** - V případě nalezení textu uvnitř elementu, je získána hodnota textu.
- **ATTRIBUT** - V případě nalezení atributu v elementu, je získán název atributu a jeho hodnota.
- **PROC_INSTRUCTION** - V případě nalezení procesní instrukce, je získán cíl instrukce jako název a instrukce jako hodnota.
- **XML_DECLARATION** - V případě nalezení deklarace XML, je získána deklarace jako hodnota.
- **COMMENT** - V případě nalezení komentáře, je získána jejich hodnota.
- **CDATA** - V případě nalezení CDat, je získána jejich hodnota.

V případě neočekávané chyby při otevírání souboru, nebo jeho čtení, může dojít k následujícím výjimkám:

- **ParserException** - V případě chyby při otevírání nebo zpracováváním souboru nesouvisejícím se syntaxí XML, nebo v případě, že parser narazí na neimplementovanou syntaxi XML.
- **XmlException** - V případě, že parser narazí na nesprávnou syntaxi XML dokumentu.

Na výpisu 17 lze vidět příklad vypsání všech názvů elementů v souboru "test.xml"pomocí třídy `XmlReader`.

```
try
{
   VsbFeiXml::XmlReader reader = VsbFeiXml::XmlReader("xmark1.xml");
    while(parser->read())
    {
      if(parser->getNodeType == XmlNodeType::START_ELEMENT) {
            std::cout << reinterpret_cast<char*>(reader.getName()->data()) <<
                std::endl;
        }
    }
}
catch (VsbFeiXml::ParserException e) {
   std::cout << e.what();
}
catch (VsbFeiXml::XmlException e) {
   std::cout << e.what();
}
```

Výpis 17: Příklad vypsání všech názvů elementů pomocí třídy XmlReader.

4.2 XmlDocument

Třída XmlDocument dokáže zpracovat XML dokument do objektově modelového dokumentu, který pak lze číst, modifikovat a uložit do souboru. Jelikož potřebuje načíst celý XML dokument do paměti, je vhodnější pro náročnější operace s menšími XML dokumenty. Ke čtení využívá třídu XmlReader. Pro použití třídy je potřeba přesunout soubory **XmlReader.h**, **XmlReader.cpp**, **XmlDocument.h** a **XmlDocument.cpp** do složky se zdrojovými soubory a přidat hlavičkový soubor, tam kde bude používán. Výpis 18 obsahuje přidání potřebného hlavičkového souboru a následné vytvoření instance třídy XmlDocument na stacku.

```
#include "XmlDocument.h"

VsbFeiXml::XmlDocument doc;
```

Výpis 18: Přidání knihovny XmlDocument.

XmlDocument využívá třídy pro každý prvek XML dokumentu:

- **XmlNode** - Abstraktní třída z které dědí všechny ostatní třídy. Obsahuje základní operace s prvky XML (viz popis níže).
- **XmlDocument** - Samotný XML dokument. Obsahuje metody pro vytváření nových prvků XML.
- **XmlElement** - Třída pro element, obsahuje metody pro správa atributů.
- **XmlAttribute** - Atribut elementu.
- **XmlText** - Text uvnitř elementu.
- **XmlComment** - Komentář.
- **CDataSection** - CData.
- **XmlDeclaration** - Deklarace XML.
- **XmlProcessingInstruction** - Procesní instrukce.

Všechny výše zmíněné třídy kromě XmlDocument mají privátní konstruktor a musí být vytvořeny pomocí následujících method volaných na instanci třídy XmlDocument:

- **createElement**() : *XmlElement** - Vytvoření nového elementu.
- **createText**() : *XmlText** - Vytvoření nového textu

- **createComment**() : *XmlComment** - Vytvoření nového komentáře
- **createCDataSection**() : *XmlCDataSection** - Vytvoření nových CDat
- **createAttribute**() : *XmlAttribute** - Vytvoření nového atributu
- **createProcessingInstruction**() : *XmlProcessingInstruction** - Vytvoření nové procesní instrukce

Pro získání informací o uzlech a pro procházení potomků elementu a dokumentu slouží následující metody ve třídě `XmlNode`:

- **getNodeType**() : *XmlNodeType** - Získání typu uzlu XML.
- **getValue**() : *char** - Získání hodnoty uzlu.
- **getName**() : *char** - Získání názvu uzlu.
- **getParent**() : *XmlNode** - Získání rodiče uzlu.
- **getFirstChild**() : *XmlNode** - Získání prvního potomka uzlu.
- **getLastChild**() : *XmlNode** - Získání posledního potomka uzlu.
- **getNextSibling**() : *XmlNode** - Získání dalšího sourozeneckého uzlu.
- **getPreviousSibling**() : *XmlNode** - Získání předchozího sourozeneckého uzlu.
- **getFirstElementChild**() : *XmlNode** - Získání prvního potomku typu element.

K přidání nových uzlů, jejich odstranění a nebo nastavení jejich názvu nebo hodnoty slouží následující metody:

- **appendChild**(child : *XmlNode**) : *XmlNode** - Přidání nového potomka na konec seznamu.
- **prependChild**(child : *XmlNode**) : *XmlNode** - Přidání nového potomka na začátek seznamu.
- **removeChild**(child : *XmlNode**) : *void** - Odstraní potomka ze seznamu.
- **removeChildren**(child : *XmlNode**) : *void* - Odstraní všechny potomky ze seznamu.
- **setName**(name : *char**) : *void* - Nastaví název uzlu.
- **setValue**(value : *char**) : *void* - Nastaví hodnotu uzlu.

Pro manipulaci s atributy jsou k dispozici v třídě XmlElement následující metody:

- **setAttribute**(name : *char**, value : *char**) : *void* - Nastaví elementu atribut. V případě že již existuje element se stejným názvem, je nahrazen novým.
- **setAttributeNode**(attribute : *XmlAttribute**) : *XmlAttribute** - Nastaví elementu atribut. V případě že již existuje element se stejným názvem, tak je nahrazen novým.
- **hasAttribute**(name : *char**) : *bool* - Vrací true, pokud má element atribut s daným jménem.
- **getAttribute**(name : *char**) : *char** - Vrací hodnotu atributu s odpovídajícím názvem. Pokud nenalezne, vrací nulový ukazatel.
- **getAttributeNode**(name : *char**) : *XmlAttribute** - Vrací atribut s odpovídajícím názvem. Pokud nenalezne, vrací nulový ukazatel.
- **getFirstAttribute**() : *XmlAttribute** - Vrací první atribut v seznamu.
- **removeAttribute**(name : *char**) : *void* - Odstraní atribut s daným názvem.
- **removeAttributeNode**(attribute : *XmlAttribute**) : *void* - Odstraní atribut s daným názvem.
- **removeAllAttributes**() : *void* - Odstraní všechny atributy.

XmlNodeType v případě této třídy může nabývat následujících hodnot:

- **ELEMENT**
- **ATTRIBUTE**
- **TEXT**
- **COMMENT**
- **CDATA**
- **PROC_INSTRUCTION**
- **XML_DECLARATION**

V případě chyby při parsování souboru, je vygenerována jedna z výjimek třídy `XmlReader`. Pokud je vyvolána neoprávněná XML operace nad dokumentem, jako je například přidání nového uzlu do komentáře, je vygenerována výjimka `InvalidOperationException`.

5 Závěr

Cílem této práce bylo důkladně vysvětlit co je to XML a důkladně popsat jeho syntaxi. XML bylo popsáno tak, aby byl každý schopný pochopit co znamená XML, čím se vyznačuje, kým byl vytvořen, jaké byly cíle při jeho návrhu, k čemu se využívá a jak se tvoří XML dokument.

Dalším cílem bylo navrhnout architekturu parseru pro rozhraní SAX a následně ho implementovat v jazyce C++ s důrazem na rychlost zpracování XML dokumentu. Třída XmlReader je schopná zpracovat XML dokumenty ze souboru s kódováním UTF-8 libovolné velikosti bez podpory DTD a schopnosti kontrolovat správnost struktury dokumentu podle schématu. Zpracování dokumentu zvládně se stejnou rychlostí jako konkurenční Xerces SAX2 parser ve stejném jazyce. Dokumenty pod 100MB dokáže zpracovat rychleji než parsery v jiných jazycích. U dokumentů o velikosti nad 1GB začne oproti konkurenci potřebuje o 50% více času pro zpracování souboru.

Posledním cílem bylo navrhnout a implementovat parser s rozhraním DOM. Třída XmlDocument dokáže pomocí třídy XmlReader zpracovat XML dokument ze souboru a načíst ho jako stromovou strukturu do operační paměti s průměrnou efektivitou 1 byte souboru na 5 bytů operační paměti. Následně je možné prostřednictvím kódu manipulovat s dokumentem. Je možné uzly přidávat, mazat nebo upravovat jejich data.

Do budoucna by bylo možné třídu XmlReader rozšířit ještě o podporu DTD, aby bylo možné kontrolovat strukturu zpracovávaného XML dokumentu. Dále je potřeba snížit čas potřebný pro zpracovávání velkých XML souborů. XmlDocument je velmi neefektivní při častém vytváření a mazání uzlů, jelikož se nyní při vytváření nových uzlů, musí pokaždé alokovat nový kus paměti. Vyřešit by to šlo použitím paměťových poolů, kdy by bylo možné recyklovat alokovanou paměť z vymazaných uzlů pro vytvoření nových.

Literatura

1. ČERBA, Otakar. *Základy XML – struktura dokumentu* [online] [cit. 2019-04-19]. Dostupné z: http://old.gis.zcu.cz/studium/pok/Materialy/02_XML_zaklady_testy.pdf.
2. BĚHÁLEK, Marek. *1. Stručný úvod do XML* [online] [cit. 2019-04-19]. Dostupné z: http://www.cs.vsb.cz/behalek/vyuka/pcsharp/text/ch08s01.html.
3. W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online] [cit. 2019-04-19]. Dostupné z: https://www.w3.org/TR/xml/.
4. *Extensible Markup Language* [online] [cit. 2019-04-19]. Dostupné z: https://cs.wikipedia.org/wiki/Extensible_Markup_Language.
5. W3C. *Extensible Markup Language (XML) 1.1 (Second Edition)* [online] [cit. 2019-04-19]. Dostupné z: https://www.w3.org/TR/xml11/.
6. *An example XML code snippet with the corresponding tree structure* [online] [cit. 2020-05-02]. Dostupné z: https://www.researchgate.net/figure/An-example-XML-code-snippet-with-the-corresponding-tree-structure_fig1_228930844.
7. KOSEK, Jiří. *Parsery* [online] [cit. 2019-04-19]. Dostupné z: https://www.kosek.cz/clanky/swn-xml/ar02s30.html.
8. *Základy XML – struktura dokumentu* [online] [cit. 2019-04-19]. Dostupné z: https://en.wikipedia.org/wiki/XML.
9. *C++* [online] [cit. 2019-04-21]. Dostupné z: https://en.wikipedia.org/wiki/C%5C%2B%5C%2B.
10. *Introduction to Visual Studio* [online] [cit. 2020-05-02]. Dostupné z: https://www.geeksforgeeks.org/introduction-to-visual-studio/.
11. *Microsoft Visual C++* [online] [cit. 2020-05-02]. Dostupné z: https://en.wikipedia.org/wiki/Microsoft_Visual_C%2B%2B.
12. *Very Sleepy* [online] [cit. 2020-05-02]. Dostupné z: http://www.codersnotes.com/sleepy/.

A Zdrojové kódy

XmlReader.h – Hlavičkový soubor knihovny XmlReader

XmlReader.cpp – Zdrojový kód knihovny XmlReader

XmlDocument.h – Hlavičkový soubor knihovny XmlDocument

XmlDocument.cpp – Zdrojový kód knihovny XmlDocument

B Tabulky srovnání s existujícími parsery

	Java SAXParser	C# XmlReader	C++ Xerces	C++ XmlReader
xmark 0.1	286	209	94	110
xmark 1	905	966	927	1041
xmark 10	6237	6190	9358	10041
SwissProt	1273	1371	1848	1406
Treebank	914	964	1073	958

Tabulka 4: Srovnání parserů s rozhraním SAX.

	C++ XmlDocument	C# XmlDocument	Java DocBuilder	C++ TinyXML-2
xmark 0.1	603	921	1213	469
xmark 1	4891	3954	4500	3371
SwissProt	6542	5426	6550	4325
Treebank	5559	2985	4104	3435

Tabulka 5: Srovnání parserů s rozhraním DOM.